Compiler Construction

Lecture 17: Code Generation III (Implementation of Static Data Structures)

Thomas Noll

Lehrstuhl für Informatik 2
(Software Modeling and Verification)

RWTH AACHEN UNIVERSITY

noll@cs.rwth-aachen.de

http://moves.rwth-aachen.de/teaching/ss-14/cc14/

Summer Semester 2014

Online Registration for Seminars and Practical Courses (Praktika) in Winter Term 2014/15

Who?

Students of:
- Master Courses
- Bachelor Informatik (~~Pro~~Seminar!)

Where?

www.graphics.rwth-aachen.de/apse

When?

04.07.2014 - 20.07.2014

Outline

Syntax of EPL

Definition (Syntax of EPL)

The syntax of EPL is defined as follows:

$$
\begin{array}{lll}
\mathbb{Z}: & z & (* \ z \text{ is an integer } *) \\
\mathit{Ide}: & I & (* \ I \text{ is an identifier } *) \\
\mathit{AExp}: & A ::= z \mid I \mid A_1 + A_2 \mid \ldots & \\
\mathit{BExp}: & B ::= A_1 < A_2 \mid \texttt{not}\ B \mid B_1\ \texttt{and}\ B_2 \mid B_1\ \texttt{or}\ B_2 & \\
\mathit{Cmd}: & C ::= I\ \texttt{:=}\ A \mid C_1\,\texttt{;}\,C_2 \mid \texttt{if}\ B\ \texttt{then}\ C_1\ \texttt{else}\ C_2 \mid & \\
 & \quad \texttt{while}\ B\ \texttt{do}\ C \mid I\,\texttt{()} & \\
\mathit{Dcl}: & D ::= D_C\ D_V\ D_P & \\
 & D_C ::= \varepsilon \mid \texttt{const}\ I_1\ \texttt{:=}\ z_1, \ldots, I_n\ \texttt{:=}\ z_n\texttt{;} & \\
 & D_V ::= \varepsilon \mid \texttt{var}\ I_1, \ldots, I_n\texttt{;} & \\
 & D_P ::= \varepsilon \mid \texttt{proc}\ I_1\texttt{;}K_1\texttt{;} \ldots \texttt{;proc}\ I_n\texttt{;}K_n\texttt{;} & \\
\mathit{Blk}: & K ::= D\ C & \\
\mathit{Pgm}: & P ::= \texttt{in/out}\ I_1, \ldots, I_n\texttt{;}K\,\texttt{.} &
\end{array}
$$

Outline

Implementation of Data Structures

Source code: data structures = arrays, records, lists, trees, ...
$\Longrightarrow$ structured state space, variables with components

Abstract machine: linear memory structure, cells for storing atomic data

Translation: mapping of structured state space to linear memory
($\Longrightarrow$ address computation)

- static data structures: memory reqirements known at compile time
- dynamic data structures: memory reqirements runtime dependent
  $\Longrightarrow$ heap, pointers, garbage collection, ...

First step:

- static data structures (arrays and records)
- inductive type definitions
- no procedures (for simplification; "orthogonal" extension)

Outline

Modified Syntax of EPL

Definition 17.1 (Modified syntax of EPL)

The modified syntax of EPL is defined as follows (where $n \geq 1$):

$$
\begin{array}{lll}
\mathbb{Z}: & z & (* \ z \text{ is an integer } *) \\
\mathbb{B}: & b ::= \texttt{true} \mid \texttt{false} & (* \ b \text{ is a Boolean } *) \\
\mathbb{R}: & r & (* \ r \text{ is a real number } *) \\
\mathit{Con}: & c ::= z \mid b \mid r & (* \ c \text{ is a constant } *) \\
\mathit{Ide}: & I & (* \ I \text{ is an identifier } *) \\
\mathit{Type}: & T ::= \texttt{bool} \mid \texttt{int} \mid \texttt{real} \mid I \mid \texttt{array}[z_1 \texttt{..} z_2] \ \texttt{of} \ T \mid & \\
 & \quad \texttt{record} \ I_1 : T_1 ; \ldots ; I_n : T_n \ \texttt{end} & \\
\mathit{Var}: & V ::= I \mid V[E] \mid V.I & \\
\mathit{Exp}: & E ::= c \mid V \mid E_1 + E_2 \mid E_1 < E_2 \mid E_1 \ \texttt{and} \ E_2 \mid \ldots & \\
\mathit{Cmd}: & C ::= V := E \mid C_1 ; C_2 \mid & \\
 & \quad \texttt{if} \ E \ \texttt{then} \ C_1 \ \texttt{else} \ C_2 \mid \texttt{while} \ E \ \texttt{do} \ C & \\
\mathit{Dcl}: & D ::= D_C \ D_T \ D_V & \\
 & D_C ::= \varepsilon \mid \texttt{const} \ I_1 := c_1 ; \ldots ; I_n := c_n ; & \\
 & D_T ::= \varepsilon \mid \texttt{type} \ I_1 := T_1 ; \ldots ; I_n := T_n ; & \\
 & D_V ::= \varepsilon \mid \texttt{var} \ I_1 : T_1 ; \ldots ; I_n : T_n ; & \\
\mathit{Pgm}: & P ::= D \ C &
\end{array}
$$

Static Semantics I

- All identifiers in a declaration D have to be different.
- In $T = \texttt{record}\ I_1 : T_1 ; \ldots ; I_n : T_n\ \texttt{end}$, all selectors I_j must be different.
- In $T = \texttt{array}[z_1 \texttt{..} z_2]\ \texttt{of}\ T$, $z_1 \leq z_2$.
- Type definitions must not be recursive:
  if $D_T = \texttt{type}\ I_1 := T_1 ; \ldots ; I_n := T_n ;$ and type identifier I occurs in T_j, then $I \in \{I_1, \ldots, I_{j-1}\}$.
- All type identifiers used in in a variable declaration D_V must be declared in D_T.
- Every identifier used in a command C must be declared in D (as a constant or variable).
- Variables in expressions and assignments have a base type (`bool`/`int`/`real`; possibly via type identifiers).

Static Semantics II

- Array indices must have type `int`.
- Execution conditions (`while`) and branching expressions (`if`) must have type `bool`.
- The types of the left-hand side and of the right-hand side types of an assignment must be compatible.
- Type compatibility: $\mathbb{Z} \subseteq \mathbb{R}$ in mathematics, but not on computers (different representation)

  $\Longrightarrow$ type casts

  weak typing: implicit casting by compiler (`2.5 + 1`, `1 + "42"`)

  $\Longrightarrow$ risc of undetected "real" errors;
  for programming-in-the-small (script languages)

  strong typing: explicit casting by programmer

  $\Longrightarrow$ enhanced software reliability;
  for programming-in-the-large
- Instantiation of operators/functions/procedures/... for different parameter types: polymorphism or overloading

  $\texttt{+} : \texttt{int} \times \texttt{int} \to \texttt{int} \qquad \texttt{+} : \texttt{real} \times \texttt{real} \to \texttt{real}$

Outline

The Modified Abstract Machine AM

- Additional main storage for keeping data values
- Procedure stack not required anymore (as procedures no longer supported)

Definition 17.2 (Modified abstract machine for EPL)

The modified abstract machine for EPL (AM) is defined by the state space

$$S := PC \times DS \times MS$$

with

- the program counter $PC := \mathbb{N}$,
- the data stack $DS := \mathbb{R}^*$ (top to the right), and
- the main storage $MS := \{\sigma \mid \sigma : \mathbb{N} \to \mathbb{R}\}$.

New AM Instructions

Definition 17.3 (New AM instructions)

- Procedure instructions are no longer needed.
- Transfer instructions ($\texttt{LOAD}(dif, off)$, $\texttt{STORE}(dif, off)$) are replaced by the following instructions with the respective semantics $[\![O]\!] : S \dashrightarrow S$:

$$
\begin{aligned}
[\![\texttt{LOAD}]\!](a, d : n, \sigma) &:= (a+1, d : \sigma(n), \sigma) \\
&\quad \text{if } n \in \mathbb{N} \\
[\![\texttt{STORE}]\!](a, d : n : r, \sigma) &:= (a+1, d, \sigma[n \mapsto r]) \\
&\quad \text{if } n \in \mathbb{N}
\end{aligned}
$$

- Moreover the following instruction for checking array bounds is introduced:

$$
[\![\texttt{CAB}(z_1, z_2)]\!](a, d : z, \sigma) := \begin{cases} (a+1, d : z, \sigma) & \text{if } z \in \{z_1, \ldots, z_2\} \\ (0, d : \underbrace{\texttt{RTE}}_{\text{runtime error}}, \sigma) & \text{otherwise} \end{cases}
$$

Outline

Modifying the Symbol Table

$$\begin{aligned} \mathit{Tab} := \{\mathrm{st} \mid \mathrm{st} : \mathit{Ide} \dashrightarrow\ & (\{\texttt{const}\} \times (\mathbb{B} \cup \mathbb{Z} \cup \mathbb{R})) \\ & \cup (\{\texttt{var}\} \times \mathit{Ide} \times \mathbb{N}) \\ & \cup (\{\texttt{type}\} \times \{\texttt{bool}, \texttt{int}, \texttt{real}\} \times \{1\}) \\ & \cup (\{\texttt{type}\} \times \{\texttt{array}\} \times \mathbb{Z}^2 \times \mathit{Ide} \times \mathbb{N}) \\ & \cup (\{\texttt{type}\} \times \{\texttt{record}\} \times (\mathit{Ide}^2 \times \mathbb{N})^* \times \mathbb{N})\} \end{aligned}$$

Remarks:

- Variable descriptor $(\texttt{var}, I, n)$: type I, memory address n
- Last component of `type` entry: memory requirement
  (base types: 1 "cell")
- Array descriptor $(\texttt{type}, \texttt{array}, z_1, z_2, I, n)$:
  bounds z_1, z_2, component type I
- Record descriptor $(\texttt{type}, \texttt{record}, I_1, J_1, o_1, \ldots, I_l, J_l, o_l, n)$: selector I_k, component type J_k, memory offset o_k
- "Indexed" table lookup: $\mathrm{st}(I.I_k) := (J_k, o_k)$
  if $\mathrm{st}(I) = (\texttt{type}, \texttt{record}, \ldots, I_k, J_k, o_k, \ldots, n)$

Maintaining the Symbol Table I

The symbol table is again maintained by the function $\mathrm{update}(D, \mathrm{st})$ which specifies the update of symbol table st according to declaration D.

For the sake of simplicity we assume that $D = D_C\ D_T\ D_V \in Dcl$ is flattened, i.e., that every subtype is named by an identifier:

- If $D_T = \texttt{type}\ \ I_1 := T_1\,; \ldots ; I_n := T_n\,;$, then for every $k \in [n]$
  - $T_k \in \{\texttt{bool}, \texttt{int}, \texttt{real}\}$ or
  - $T_k \in \{I_1, \ldots, I_{k-1}\}$ or
  - $T_k = \texttt{array}\,[z_1 \,..\, z_2]\ \ \texttt{of}\ \ I_j$ where $j \in [k-1]$ or
  - $T_k = \texttt{record}\ \ J_1 : I_{j_1}\,; \ldots ; J_l : I_{j_l}\ \ \texttt{end}$ where $j_1, \ldots, j_l \in [k-1]$
- For D_T as above, D_V must be of the form $D_V = \texttt{var}\ \ J_1 : I_{j_1}\,; \ldots ; J_k : I_{j_k}\,;$ where $j_1, \ldots, j_k \in [n]$

Maintaining the Symbol Table II

Definition 17.1 (Modified update function)

update : $Dcl \times Tab \dashrightarrow Tab$ is defined by

$$\begin{aligned}
\text{update}(D_C\ D_T\ D_V, \text{st}) &:= \text{update}(D_V, \text{update}(D_T, \text{update}(D_C, \text{st}))) \\
\text{update}(\varepsilon, \text{st}) &:= \text{st} \\
\text{update}(\texttt{const}\ I_1 \texttt{:=} c_1 \texttt{;} \ldots \texttt{;} I_n \texttt{:=} c_n \texttt{;}, \text{st}) & \\
&:= \text{st}[I_1 \mapsto (\texttt{const}, c_1), \ldots, I_n \mapsto (\texttt{const}, c_n)] \\
\text{update}(\texttt{type}\ I \texttt{:=bool;} D'_T, \text{st}) &:= \text{update}(\texttt{type}\ D'_T, \text{st}[I \mapsto (\texttt{type}, \texttt{bool}, 1)]) \\
\text{update}(\texttt{type}\ I \texttt{:=int;} D'_T, \text{st}) &:= \text{update}(\texttt{type}\ D'_T, \text{st}[I \mapsto (\texttt{type}, \texttt{int}, 1)]) \\
\text{update}(\texttt{type}\ I \texttt{:=real;} D'_T, \text{st}) &:= \text{update}(\texttt{type}\ D'_T, \text{st}[I \mapsto (\texttt{type}, \texttt{real}, 1)]) \\
\text{update}(\texttt{type}\ I \texttt{:=} J \texttt{;} D'_T, \text{st}) &:= \text{update}(\texttt{type}\ D'_T, \text{st}[I \mapsto \text{st}(J)]) \\
\text{update}(\texttt{type}\ I \texttt{:=array[} z_1 \texttt{..} z_2 \texttt{]}\ \texttt{of}\ J \texttt{;} D'_T, \text{st}) & \\
&:= \text{update}(\texttt{type}\ D'_T, \\
&\quad \text{st}[I \mapsto (\texttt{type}, \texttt{array}, z_1, z_2, J, k \cdot n)]) \\
&\quad \text{if } \text{st}(J) = (\texttt{type}, \ldots, n) \text{ and } k = z_2 - z_1 + 1 \\
\text{update}(\texttt{type}\ I \texttt{:=record}\ I_1 \texttt{:} J_1 \texttt{;} \ldots \texttt{;} I_l \texttt{:} J_l\ \texttt{end;} D'_T, \text{st}) & \\
&:= \text{update}(\texttt{type}\ D'_T, \text{st}[I \mapsto \\
&\quad (\texttt{type}, \texttt{record}, I_1, J_1, 0, I_2, J_2, n_1, \ldots, \\
&\quad I_l, J_l, \textstyle\sum_{i=1}^{l-1} n_i, \sum_{i=1}^{l} n_i)]) \\
&\quad \text{if } \text{st}(J_i) = (\texttt{type}, \ldots, n_i) \text{ for } i \in [l] \\
\text{update}(\texttt{var}\ I_1 \texttt{:} J_1 \texttt{;} \ldots \texttt{;} I_n \texttt{:} J_n \texttt{;}, \text{st}) &:= \text{st}[I_1 \mapsto (\texttt{var}, J_1, 0), I_2 \mapsto (\texttt{var}, J_2, n_1), \ldots, \\
&\quad I_n \mapsto (\texttt{var}, J_n, \textstyle\sum_{i=1}^{n-1} n_i)] \\
&\quad \text{if } \text{st}(J_i) = (\texttt{type}, \ldots, n_i) \text{ for } i \in [l]
\end{aligned}$$

Maintaining the Symbol Table III

Example 17.2 (Modified update function)

Let $D :=$
```
type Bool=bool; Int=int;
     Array=array[1..20] of Bool;
     Record=record S:Array; T:Int end;
var x:Int; y:Array; z:Record;
```

Then

$$\begin{aligned}\mathrm{update}(D, \mathrm{st}) = \mathrm{st}[\ & \texttt{Bool} \mapsto (\texttt{type}, \texttt{bool}, 1),\\ & \texttt{Int} \mapsto (\texttt{type}, \texttt{int}, 1),\\ & \texttt{Array} \mapsto (\texttt{type}, \texttt{array}, 1, 20, \texttt{Bool}, 20),\\ & \texttt{Record} \mapsto (\texttt{type}, \texttt{record}, \texttt{S}, \texttt{Array}, 0, \texttt{T}, \texttt{Int}, 20, 21),\\ & \texttt{x} \mapsto (\texttt{var}, \texttt{Int}, 0),\\ & \texttt{y} \mapsto (\texttt{var}, \texttt{Array}, 1),\\ & \texttt{z} \mapsto (\texttt{var}, \texttt{Record}, 21)]\end{aligned}$$

Translation of Variables I

The translation employs the following auxiliary function to determine the type identifier of a given variable:

Definition 17.3 (vtype function)

The mapping

$$\mathrm{vtype} : \mathit{Var} \times \mathit{Tab} \dashrightarrow \mathit{Ide}$$

is given by

$$\begin{aligned}
\mathrm{vtype}(I, \mathrm{st}) &:= J \\
&\quad \text{if } \mathrm{st}(I) = (\mathtt{var}, J, n) \\
\mathrm{vtype}(V\,[E], \mathrm{st}) &:= J \\
&\quad \text{if } \mathrm{vtype}(V, \mathrm{st}) = I \\
&\quad \text{and } \mathrm{st}(I) = (\mathtt{type}, \mathtt{array}, z_1, z_2, J, n) \\
\mathrm{vtype}(V\,.\,I, \mathrm{st}) &:= J \\
&\quad \text{if } \mathrm{vtype}(V, \mathrm{st}) = I' \text{ and } \mathrm{st}(I'.I) = (J, o)
\end{aligned}$$

Translation of Variables II

The function vt generates code for computing the memory address of a variable (and storing it on the data stack):

Definition 17.4 (Translation of variables)

The mapping

$$\mathrm{vt} : \mathit{Var} \times \mathit{Tab} \dashrightarrow \mathit{AM}$$

is given by

$$\begin{aligned}
\mathrm{vt}(I, \mathrm{st}) &:= \texttt{LIT}(n)\texttt{;} \\
&\quad \text{if } \mathrm{st}(I) = (\texttt{var}, J, n) \\
\mathrm{vt}(V\texttt{[}E\texttt{]}, \mathrm{st}) &:= \mathrm{vt}(V, \mathrm{st}) && \text{\% address of } V \\
&\quad \mathrm{et}(E, \mathrm{st}) && \text{\% array index} \\
&\quad \texttt{CAB}(z_1\texttt{,}z_2)\texttt{;} && \text{\% bounds checking} \\
&\quad \texttt{LIT}(z_1)\texttt{; SUB;} && \text{\% index difference} \\
&\quad \texttt{LIT}(n)\texttt{; MULT;} && \text{\% relative address} \\
&\quad \texttt{ADD;} && \text{\% address of } V\texttt{[}E\texttt{]} \\
&\quad \text{if } \mathrm{vtype}(V, \mathrm{st}) = I \text{ and } \mathrm{st}(I) = (\texttt{type}, \texttt{array}, z_1, z_2, J, m) \\
&\quad \text{and } \mathrm{st}(J) = (\texttt{type}, \ldots, n) \\
\mathrm{vt}(V\texttt{.}I, \mathrm{st}) &:= \mathrm{vt}(V, \mathrm{st}) && \text{\% address of } V \\
&\quad \texttt{LIT}(o)\texttt{;} && \text{\% offset} \\
&\quad \texttt{ADD;} && \text{\% address of } V\texttt{.}I \\
&\quad \text{if } \mathrm{vtype}(V, \mathrm{st}) = I' \text{ and } \mathrm{st}(I'\texttt{.}I) = (J, o)
\end{aligned}$$

Translation of Expressions

Definition 17.5 (Translation of expressions)

The mapping

$$\mathrm{et} : \mathit{Exp} \times \mathit{Tab} \dashrightarrow \mathit{AM}$$

is given by

$$\begin{aligned}
\mathrm{et}(c, \mathrm{st}) &:= \texttt{LIT(}c\texttt{);} \\
\mathrm{et}(V, \mathrm{st}) &:= \begin{cases} \texttt{LIT(}c\texttt{);} & \text{if } V \in \mathit{Ide} \text{ and } \mathrm{st}(V) = (\texttt{const}, c) \\ \mathrm{vt}(V, \mathrm{st}) & \text{otherwise} \\ \texttt{LOAD;} \end{cases} \\
\mathrm{et}(E_1 \texttt{+} E_2, \mathrm{st}) &:= \mathrm{et}(E_1, \mathrm{st}) \\
&\quad\ \mathrm{et}(E_2, \mathrm{st}) \\
&\quad\ \texttt{ADD;} \\
&\ \ \vdots
\end{aligned}$$

Translation of Commands and Programs

Definition 17.6 (Translation of commands)

For the mapping

$$\mathrm{ct} : \mathit{Cmd} \times \mathit{Tab} \dashrightarrow \mathit{AM}$$

only the handling of assignments needs to be adapted:

$$\begin{aligned} \mathrm{ct}(V\,\texttt{:=}\,E, \mathrm{st}) := {} & \mathrm{vt}(V, \mathrm{st}) && \%\ \text{address of left-hand side} \\ & \mathrm{et}(E, \mathrm{st}) && \%\ \text{value of right-hand side} \\ & \texttt{STORE;} \end{aligned}$$

Definition 17.7 (Translation of programs)

The mapping

$$\mathrm{trans} : \mathit{Pgm} \dashrightarrow \mathit{AM}$$

is defined by

$$\mathrm{trans}(D\ C) := \mathrm{ct}(C, \mathrm{update}(D, \mathrm{st}_\emptyset))$$

Outline

Translation Example I

Example 17.8

$P =$ `type Int=int; Array=array[1..10] of Int;` } D
`var a:Array; i:Int;`
`i:=1;` } C
`while i<=10 do`
`a[i]:=i; i:=i+1;`

$$
\begin{aligned}
\text{trans}(P) &= \text{ct}(C, \text{update}(D, \text{st}_\emptyset)) \\
\text{st} &:= \text{update}(D, \text{st}_\emptyset) \\
&= \text{st}_\emptyset[\ \texttt{Int} \mapsto (\texttt{type}, \texttt{int}, 1), \\
&\qquad \texttt{Array} \mapsto (\texttt{type}, \texttt{array}, 1, 10, \texttt{Int}, 10), \\
&\qquad \texttt{a} \mapsto (\texttt{var}, \texttt{Array}, 0), \\
&\qquad \texttt{i} \mapsto (\texttt{var}, \texttt{Int}, 10)] \\
\text{ct}(C, \text{st}) &= \text{ct}(\texttt{i:=1}, \text{st}) \\
&\quad\ \text{ct}(\texttt{while i<=10 do a[i]:=i; i:=i+1}, \text{st}) \\
\text{ct}(\texttt{i:=1}, \text{st}) &= \text{vt}(\texttt{i}, \text{st}) \quad \% \ \text{adr(i)} \\
&\quad\ \text{et}(1, \text{st}) \quad \% \ \text{val(1)} \\
&\quad\ \texttt{STORE;} \\
&= \texttt{LIT(10); LIT(1); STORE;}
\end{aligned}
$$

Translation Example II

Example 17.8 (continued)

$$
\begin{aligned}
&\mathrm{ct}(\texttt{while i<=10 do a[i]:=i; i:=i+1}, \mathrm{st}) \\
&\qquad = a:\ \mathrm{et}(\texttt{i<=10}, \mathrm{st}) \\
&\qquad\quad\ \texttt{JFALSE}(a'); \\
&\qquad\quad\ \mathrm{ct}(\texttt{a[i]:=i; i:=i+1}, \mathrm{st}) \\
&\qquad\quad\ \texttt{JMP}(a); \\
&\qquad\ a':\\
&\mathrm{et}(\texttt{i<=10}, \mathrm{st}) = \texttt{LIT(10); LOAD; LIT(10); LE;} \\
&\mathrm{ct}(\texttt{a[i]:=i; i:=i+1}, \mathrm{st}) = \mathrm{ct}(\texttt{a[i]:=i}, \mathrm{st})\ \mathrm{ct}(\texttt{i:=i+1}, \mathrm{st}) \\
&\mathrm{ct}(\texttt{a[i]:=i}, \mathrm{st}) = \mathrm{vt}(\texttt{a[i]}, \mathrm{st}) \quad \%\ \mathrm{adr}(\texttt{a[i]}) \\
&\qquad\qquad\qquad\ \mathrm{et}(\texttt{i}, \mathrm{st}) \quad\ \%\ \mathrm{val}(\texttt{i}) \\
&\qquad\qquad\qquad\ \texttt{STORE;} \\
&\mathrm{vt}(\texttt{a[i]}, \mathrm{st}) = \mathrm{vt}(\texttt{a}, \mathrm{st}) \quad\ \%\ \mathrm{adr}(\texttt{a}) \\
&\qquad\qquad\qquad\ \mathrm{et}(\texttt{i}, \mathrm{st}) \quad\ \%\ \mathrm{val}(\texttt{i}) \\
&\qquad\qquad\qquad\ \texttt{CAB(1,10);} \quad\ \%\ \text{bounds checking} \\
&\qquad\qquad\qquad\ \texttt{LIT(1); SUB;} \quad\ \%\ \text{index diff.} \\
&\qquad\qquad\qquad\ \texttt{LIT(1); MULT;} \quad\ \%\ \text{rel. address} \\
&\qquad\qquad\qquad\ \texttt{ADD;} \quad\ \%\ \mathrm{adr}(\texttt{a[i]}) \\
&\mathrm{vt}(\texttt{a}, \mathrm{st}) = \texttt{LIT(0);} \\
&\mathrm{et}(\texttt{i}, \mathrm{st}) = \texttt{LIT(10); LOAD;} \\
&\mathrm{ct}(\texttt{i:=i+1}, \mathrm{st}) = \texttt{LIT(10); LIT(10); LOAD; LIT(1); ADD; STORE;}
\end{aligned}
$$